SOFTWARE PACKAGE MANAGEMENT

CROSS REFERENCE TO RELATED APPLICATIONS

This application is a continuation of Forbes et al., U.S. Patent Application No. 09/099,570, filed June 19, 1998, entitled, "Software Package Management," which is hereby incorporated herein by reference.

This application is related to U.S. Patent Application No. 08/764,040, entitled AUTOMATIC SOFTWARE DOWNLOADING FROM A COMPUTER NETWORK, filed on December 12, 1996, and assigned to the assignee of the present application.

COPYRIGHT NOTICE/PERMISSION

FIELD OF THE INVENTION

This invention relates generally to software distribution, and more particularly to the management of software packages after distribution.

BACKGROUND OF THE INVENTION

Historically, the primary medium for software distribution has been either the traditional floppy disk or the more recent compact disc (CD-ROM). However, more and more individuals are acquiring software by downloading it from remote server computers connected to the client computers through the Internet. Additionally, companies and organizations are distributing software to their users across their local area networks. The physical medium is the network cable itself and the supporting communication hardware, a fixed cost associated with the establishment of the network. Therefore, distributing and installing software over an existing network bypasses the cost overhead of producing CDs or floppy disks.

In addition, using the network as the distribution medium profoundly reduces the software's total cost of ownership to an extent that cannot be achieved by CDs or floppies even when the media cost almost nothing to manufacture. Software distribution via CDs and floppies obey the "pull" paradigm, where every action is user-initiated. Distribution over the network has the ability to apply a "push" paradigm which provides three main benefits.

First, the installation is "hands-free" in that the user does not have to manually install the software. Second, the software can be easily and timely upgraded from a designated location because the burden of upgrading is borne by the software itself. Third, because different types of computer hardware and operating systems can connect to a common network, software distributed over the network can be made to work across platforms or intelligent so that only the correct version of platform-specific software is pushed down to the user.

However, current methods of software distribution over a network do not fully exploit

the benefits. Existing distribution of platform-specific, or "native code," software relies on installation file formats that are hard to create, not extensible, and specific to a particular operating system. Although most current software is written in modules, there is no current mechanism that handles the situation where one component in a software program requires the presence of another to operate. If a user downloads software from a Web page, the user may discover that the program requires an external library which necessitates another network session to download, assuming the user can find the right location, and then the user must manually install the library before installing the software.

Software programs written in the popular platform-independent Java language require that the Java classes be "packaged" for distribution but the package does not contain persistent information so once Java software is installed on a client computer, all information about it is lost. It is impossible to tell what the version number is, where it came from, or whom the author is. Additionally, the current network distribution methods make it difficult to digitally sign a Java package for security purposes.

More problems arise when a user wants to execute an application which depends on both native code components and Java components since the distribution methods are completely different. Finally, once the software is downloaded and successfully installed on the client computer, no mechanism exists to track all of the components so that older versions can be easily superceded when newer version are available or that all the related components can be readily uninstalled when necessary.

Therefore, there is a need for a software distribution and tracking mechanism that handles cross-platform software, specifies the component dependencies, and is applicable to

both the older distribution media as well as to the network distribution paradigm.

SUMMARY OF THE INVENTION

The above-mentioned shortcomings, disadvantages and problems are addressed by the present invention, which will be understood by reading and studying the following specification.

A software package manager uses a distribution unit containing components for a software package and a manifest file that describes the distribution unit to manage the installation, execution, and uninstallation of software packages on a computer. For installation, the package manager acquires the manifest file and parses it to learn if the software package depends on any additional components. The package manager resolves any dependencies by acquiring a distribution unit containing the needed component and installs the dependency's distribution unit as described below. Because dependencies can nested within dependencies, the package manager recursively processes all the dependencies before finishing the installation of the software package that depends upon the additional components.

The software package manager acquires the distribution unit and extracts the components in the distribution unit into a directory on the computer. The package manager causes the operating system of the computer to install the software. The package manager then updates a code store data structure with information in the manifest file. The fields in the code store data structure contains such information as the name and version of the distribution unit, a list of the components and their location on the computer, and the source of the

distribution unit. Additional fields in the code store data structure can also contain a component version, a component data type, and a digital signature if one was affixed to the distribution unit.

During the installation, the package manager can optionally scan the code store data structure to determine if a component to be installed already exists on the computer and updates the code store data structure with the location of the later version of the component.

When a user requests execution of software, the package manager uses the code store data structure to locate the appropriate components for the operating system to use. When the user requests the uninstallation of a software package, the package manager deletes the appropriate components from the computer and updates the code store data structure accordingly.

The manifest file and distribution unit optionally are combined into a distribution unit file.

The manifest file format is common across all types of code and operating systems and easily extended to embrace new code types are they arise. The manifest file and distribution unit can be stored on all types of media from traditional magnetic and optical disks to networked servers. The distribution units for dependencies do not have to reside on the same type of media as the distribution unit or the manifest file that refers to the dependency. More than one distribution unit can be resident in a distribution unit file and a distribution unit file can contain a mixture of distribution units containing different code types.

Thus, the software package manager, the manifest file, the distribution unit and the code store data structure of the present invention solve the problems with existing distribution

mechanisms. The manifest file is not particular to a particular code type or operating system and allows for the specification of nested software dependencies. Because the manifest file contains the location of the distribution units for any dependencies, the software package manager can acquire and install the dependencies without requiring manual intervention by the user. Different types of distribution units can be mixed in a distribution unit file so that a single mechanism is used to acquire and install all types of code.

The code store data structure maintained by the software package manage contains information about the installed software such as version and installation location, and is used to resolve version discrepancies among software programs that share components. The code store data structure is used by the package manager to locate necessary component when the software is executed so that a component stored in one directory can be readily shared by software programs with components in different directories. Finally, the code store data structure eases the uninstallation process by centralizing all the information about installed components.

The present invention describes systems, clients, servers, methods, and computer-readable media of varying scope. In addition to the aspects and advantages of the present invention described in this summary, further aspects and advantages of the invention will become apparent by reference to the drawings and by reading the detailed description that follows.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows a diagram of the hardware and operating environment in conjunction

with which embodiments of the invention may be practiced;

FIGs. 2A, 2B and 2C are diagrams illustrating a system-level overview of an exemplary embodiment of a package manager of the invention;

FIGs. 3A, 3B, 3C and 3D are flowchart of methods to be performed by a client according to an exemplary embodiment of the package manager of the invention; and

FIG. 4 is a diagram of an exemplary embodiment of an entry in a code store data structure suitable for use by the methods shown in FIGs. 3A, 3B and 3C.

DETAILED DESCRIPTION OF THE INVENTION

In the following detailed description of exemplary embodiments of the invention, reference is made to the accompanying drawings which form a part hereof, and in which is shown by way of illustration specific exemplary embodiments in which the invention may be practiced. These embodiments are described in sufficient detail to enable those skilled in the art to practice the invention, and it is to be understood that other embodiments may be utilized and that logical, mechanical, electrical and other changes may be made without departing from the spirit or scope of the present invention. The following detailed description is, therefore, not to be taken in a limiting sense, and the scope of the present invention is defined only by the appended claims.

The detailed description is divided into five sections. In the first section, the hardware and the operating environment in conjunction with which embodiments of the invention may be practiced are described. In the second section, a system level overview of the invention is presented. In the third section, methods for an exemplary embodiment of the invention are

provided. In the fourth section, a particular Open Software Description implementation of the invention is described. Finally, in the fifth section, a conclusion of the detailed description is provided.

Hardware and Operating Environment

FIG. 1 is a diagram of the hardware and operating environment in conjunction with which embodiments of the invention may be practiced. The description of FIG. 1 is intended to provide a brief, general description of suitable computer hardware and a suitable computing environment in conjunction with which the invention may be implemented. Although not required, the invention is described in the general context of computer-executable instructions, such as program modules, being executed by a computer, such as a personal computer. Generally, program modules include routines, programs, objects, components, data structures, etc., that perform particular tasks or implement particular abstract data types.

Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including hand-held devices, multiprocessor systems, microprocessor-based or programmable consumer electronics, network PCs, minicomputers, mainframe computers, and the like. The invention may also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

The exemplary hardware and operating environment of FIG. 1 for implementing the invention includes a general purpose computing device in the form of a computer 20,

including a processing unit 21, a system memory 22, and a system bus 23 that operatively couples various system components, including the system memory 22, to the processing unit 21. There may be only one or there may be more than one processing unit 21, such that the processor of computer 20 comprises a single central-processing unit (CPU), or a plurality of processing units, commonly referred to as a parallel processing environment. The computer 20 may be a conventional computer, a distributed computer, or any other type of computer; the invention is not so limited.

The system bus 23 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. The system memory may also be referred to as simply the memory, and includes read only memory (ROM) 24 and random access memory (RAM) 25. A basic input/output system (BIOS) 26, containing the basic routines that help to transfer information between elements within the computer 20, such as during start-up, is stored in ROM 24. The computer 20 further includes a hard disk drive 27 for reading from and writing to a hard disk, not shown, a magnetic disk drive 28 for reading from or writing to a removable magnetic disk 29, and an optical disk drive 30 for reading from or writing to a removable optical disk 31 such as a CD ROM or other optical media.

The hard disk drive 27, magnetic disk drive 28, and optical disk drive 30 are connected to the system bus 23 by a hard disk drive interface 32, a magnetic disk drive interface 33, and an optical disk drive interface 34, respectively. The drives and their associated computer-readable media provide nonvolatile storage of computer-readable instructions, data structures, program modules and other data for the computer 20. It should be appreciated by those

skilled in the art that any type of computer-readable media which can store data that is accessible by a computer, such as magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, random access memories (RAMs), read only memories (ROMs), and the like, may be used in the exemplary operating environment.

A number of program modules may be stored on the hard disk, magnetic disk 29, optical disk 31, ROM 24, or RAM 25, including an operating system 35, one or more application programs 36, other program modules 37, and program data 38. A user may enter commands and information into the personal computer 20 through input devices such as a keyboard 40 and pointing device 42. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 21 through a serial port interface 46 that is coupled to the system bus, but may be connected by other interfaces, such as a parallel port, game port, or a universal serial bus (USB). A monitor 47 or other type of display device is also connected to the system bus 23 via an interface, such as a video adapter 48. In addition to the monitor, computers typically include other peripheral output devices (not shown), such as speakers and printers.

The computer 20 may operate in a networked environment using logical connections to one or more remote computers, such as remote computer 49. These logical connections are achieved by a communication device coupled to or a part of the computer 20; the invention is not limited to a particular type of communications device. The remote computer 49 may be another computer, a server, a router, a network PC, a client, a peer device or other common network node, and typically includes many or all of the elements described above relative to

the computer 20, although only a memory storage device 50 has been illustrated in FIG. 1. The logical connections depicted in FIG. 1 include a local-area network (LAN) 51 and a wide-area network (WAN) 52. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet.

When used in a LAN-networking environment, the computer 20 is connected to the local network 51 through a network interface or adapter 53, which is one type of communications device. When used in a WAN-networking environment, the computer 20 typically includes a modem 54, a type of communications device, or any other type of communications device for establishing communications over the wide area network 52, such as the Internet. The modem 54, which may be internal or external, is connected to the system bus 23 via the serial port interface 46. In a networked environment, program modules depicted relative to the personal computer 20, or portions thereof, may be stored in the remote memory storage device. It is appreciated that the network connections shown are exemplary and other means of and communications devices for establishing a communications link between the computers may be used.

The hardware and operating environment in conjunction with which embodiments of the invention may be practiced has been described. The computer in conjunction with which embodiments of the invention may be practiced may be a conventional computer, a distributed computer, or any other type of computer; the invention is not so limited. Such a computer typically includes one or more processing units as its processor, and a computer-readable medium such as a memory. The computer may also include a communications device such as

a network adapter or a modem, so that it is able to communicatively couple to other computers.

System Level Overview

A system level overview of the operation of an exemplary embodiment of the invention is described by reference to FIGs. 2A, 2B and 2C. The exemplary embodiment is implemented in an wide-area networking environment 52 having a server computer, such as remote computer 49 and a user or client computer, such as local computer 20, all of which are shown in FIG. 1 and described in the previous section.

Fred's Software Company has written a software package named "CoolestApp" that runs as a "plug-in application" in a World Wide Web browser, such as Microsoft Internet Explorer 4. A plug-in application is often employed to provide additional capabilities, such as multimedia or interactive controls, to browsers. One type of control application is that written to conform with Microsoft's ActiveX specifications. The plug-ins are usually written in object-oriented languages such as C++ or Java and are typically used on Web pages. The user may be prompted to download the plug-in or it may be automatically downloaded when needed.

Referring to Fig. 2A, Fred's Software Company wants to distribute the CoolestApp over the Internet from Fred's Software Company's Web server 201 to a user's computer 203. Fred's Software Company logically groups the components for the CoolestApp together into a "distribution unit" 209. The components can include platform-specific compiled binary files such as dynamic linking library (.dll) files used by the Microsoft Windows family of operating systems, Java bytecode (.class) files, or files that contain optional installation instructions for

how to use certain components contained in the distribution unit, for example, ActiveX controls may need to be registered before use. The distribution unit 209 can be a separate file or can be a portion of a "distribution unit file" 205 as explained below.

Fred's Software Company also creates a "manifest" file 207 describing the CoolestApp. The CoolestApp manifest file 207 contains information about CoolestApp, including the name of the CoolestApp distribution unit 209, the version number of the software package (all components in the distribution unit 209 have the same version number in this embodiment), and the operating systems under which the CoolestApp executes. Fred's Software Company bundles the CoolestApp distribution unit 209 and manifest file 207 into a distribution unit file 205 for storage on the server 201.

The names of other files in the distribution unit file 205, such as a text file containing licensing information or a "readme" file containing special instructions for the software package, are listed in the manifest file 207. The manifest file 207 also contains entries for software that is required to run CoolestApp but which is not included in the distribution unit file 205. Such required software represent "dependencies" and frequently include such items as language libraries and common object class libraries. A dependency can also be another software package. The manifest file 207 provides the ability to describe the software dependencies in a recursive tree format, also known as a "directed graph."

In the present example, CoolestApp is an enhanced version of a software program named "CoolApp" previously distributed by Fred's Software Company. Rather than completely rewriting CoolestApp, Fred's Software Company used the CoolApp components as a base and created additional components for the new features in CoolestApp. In the

interest of minimizing download time, Fred's Software Company does not include the original components for CoolApp in the CoolestApp distribution unit 205. Instead Fred's Software Company inserts a dependency entry in the manifest file 205 which directs a user's browser to the location on Fred's Software Company server 201 holding the distribution unit file 215 for CoolApp as illustrated in FIG. 2B.

The browser begins the installation of the CoolestApp software package to the local computer by downloading the CoolestApp distribution unit file 205. A software package manager 211 running in the underlying operating system on the user's computer 203 extracts the manifest file 207 from the distribution unit file 209 and accesses an installed package database 213 to determine that Fred's Software Company's CoolestApp is not already installed. The dependency entry in CoolestApp manifest file 207 alerts the package manager 211 that the CoolestApp depends on Fred's Software Company's CoolApp. The package manager 211 determines that CoolApp has not been previously installed and directs the browser to download the CoolApp distribution unit file 215 from the server location specified in the dependency entry.

Once the CoolApp distribution unit file 215 has been downloaded to the user's computer 203, the package manager extracts the CoolApp manifest file 217 and determines that CoolApp does not have any dependencies. The package manager 211 creates a private directory 221 for Fred's Software Company applications, named FSC, extracts the CoolApp components from the distribution unit 219 into the FSC directory, and calls the underlying operating system installation facility to install the CoolApp components. The package manager 211 registers the CoolApp components in the installed package database 213 when

the installation is successful.

Referring to FIG. 2C, the package manager 213 extracts the CoolestApp components from the CoolestApp distribution unit 209 to the FSC directory 221, calls the installation facility, and registers the CoolestApp components in the installed package database 213. The browser now can run the CoolestApp helper application from the FSC directory 221.

If the user downloads additional Fred's Software Company applications that depend upon the components in either CoolApp or CoolestApp, the package manager 211 will use the already installed components to satisfy any dependencies that reference installed software package unless the additional applications require versions later than that installed.

If after running the CoolestApp helper application, the user decides that CoolestApp is not needed, the user employs the underlying operating systems uninstall facility to uninstall CoolestApp. The uninstall facility invokes the package manager 211 which determines if the CoolApp and CoolestApp components are being used by other applications and deletes the software packages from the FSC directory 221 if not. The package manager 211 also deletes the package entries from the installed package database 213 when the packages have been deleted from the FSC directory 221.

The system level overview of the operation of an exemplary embodiment of the invention has been described in this section of the detailed description. The package manager and its supporting files have been described in relation to installing a software package having a single dependency. While the invention is not limited to any particular distribution media, for sake of clarity a simplified version of Internet software distribution has been described.

Methods of an Exemplary Embodiment of the Invention

In the previous section, a system level overview of the operation of an exemplary embodiment of the invention was described. In this section, the particular methods performed by a client or local computer of such an exemplary embodiment are described by reference to a series of flowcharts. The methods to be performed by the client computer constitute computer programs made up of computer-executable instructions. Describing the methods by reference to a flowchart enables one skilled in the art to develop such programs including such instructions to carry out the methods on suitable computerized clients (the processor of the clients executing the instructions from computer-readable media).

The software package manager of the present invention is described as providing three major functions to the runtime environment of the local computer on which it runs as illustrated in FIGs. 3A-3D. It manages the installation of software packages, it locates necessary components when software is executed, and it supports the uninstallation of software. The package manager uses the installed package database, also called a "code store" data structure, to track components of software packages that have been installed on the local computer. One of skill in the art will, upon reading the following description of the code store data structure, recognize that any type of organized data structure, including various type of data bases, is suitable for use with the package manager.

As in the exemplary embodiment described in the previous section, the manifest file contains dependency entries specifying locations of distribution units containing required software components. The distribution unit file is suitable for distributing software packages on traditional media, such as CD-ROM or floppy disk, as well as over a wide area network,

such as the Internet. The package manager extracts the manifest file and the distribution unit from the distribution unit file. In an alternate embodiment, the distribution unit and the manifest file can be stored separately on a network and the manifest file contains the network location of its corresponding distribution unit.

Installation

Referring first to FIGs. 3A and 3B, a flowchart of methods to be performed by a client according to an exemplary embodiment of the invention when installing new software is shown. This method is inclusive of the steps or acts required to be taken by the package manager.

When the distribution unit file for a software package is loaded onto a computer for installation, the software package manager running in the computer acquires the manifest file for processing (step 301). In an embodiment in which the manifest file is distributed in a distribution unit file, the package manager acquires the distribution unit file and extracts the manifest file from the distribution unit file. The package manager checks the name and version of the software package contained in the manifest file against the code store data structure to determine if the software package has already been installed (step 303). If so, the package manager exits (step 321)

If the software package is not installed, the package manager checks the manifest file to determine if the software package requires the installation of other software components (dependencies) not supplied as part of the distribution file unit (step 305) before installing the software package from the distribution unit. Such is frequently the case when a software

package is written in a common programming language such as Java or C++ which depend on object class or language libraries being present.

If there are dependencies, the package manager checks the code store data structure to determine if the dependencies are installed on the local computer (step 327). If not, the package manager uses information stored in the manifest file about the dependencies to locate a source for the dependencies. The package manager than acquires a dependency from the source specified in the manifest file (step 329). In one embodiment, the source is a remote server designated by a uniform resource locator (URL) path name. In an alternate embodiment, the source is a server on a local area network and the path name is a network drive.

After acquiring the dependency from the source, the package manager installs the dependent software components on the local computer. The installation of the dependent components is identical to the installation process for the original software package which will be described in detail below in conjunction with steps 307 through 325. Because each dependency can itself include dependencies, the package manager will install all the nested dependencies prior to finishing the installation of the original software package.

Once all the dependencies are installed, the package manager determines if a directory for the software package exists (step 307) and creates one if it does not (step 309). The package manager assigns the directory a unique name that has a high probability of being different for every computer on which the software package is installed. In one embodiment, the unique name is generated using a standard hashing algorithm having the software package name as input.

The package manager extracts the components for the software package from the distribution unit file into the directory (step 311). In one embodiment, the components are gathered into an uncompressed "zip" formatted data file which contains a directory of the files within it. Other embodiments which use similar file structures to group the components together with a directory structure will be readily apparent to one skilled in the art. The package manager then invokes the installation facility provided by the operating system to install the components (step 313).

When the installation is successful (step 315), the package manager updates the code store data structure with the information contained in the manifest file (step 317 and FIG. 3B).

The package manager creates an entry in the code store data structure for the software package that includes the name of the software package, the version number, and the name of the directory (step 331). The names of the components in the software package are stored in the corresponding software package entry in code store data structure (step 333).

In one alternate embodiment shown in FIG. 3B, as part of the update process for the code store data structure, the package manager scans the code store data structure to determine if any of the components in the software package are shared with other software packages registered in the code store data structure (step 335). If so, the package manager performs one of three basic actions depending on the version information stored in the code store data base entries for the component (step 337)

If the newly stored component is a newer version of the previously stored component (step 339) and the code store data structure entry for the previously installed software package that references the older version does not indicate it is version dependent (step 341), the

package manager removes the older version from the directory in which it is stored (step 343) and updates the code store data structure entry for the previously installed software package to point to the newer version (step 345). If there is a version dependency noted, the package manager leaves the older version in the directory (step 347).

If the newly stored component is older that the previously stored component (step 349) and the software package does not indicate a version dependency (step 351), the package manager removes the older version from the newly created directory (step 353) and updates the code store data structure entry for the newly installed software package to point to the newer version (step 355). As before, if there is a version dependency noted, the package manager does nothing to the older version (step 347).

If the components are the same version, the package manager chooses one to remove from its directory (step 359) and updates the corresponding entry to point to the other component (step 361).

In the embodiment in which the components are stored in an uncompressed zip file, or the like, the package manager uses the file directory to find the component to be deleted within the file in the appropriate directory. The package manager can, alternately, actually delete the component from the file and update the file directory or mark the component entry in the file directory as deleted depending on the particular file structure employed to hold the components.

In an alternate embodiment, the package manager does not attempt to determine if there are mismatched versions installed and each software package uses the version of the component that is indicated by its entry in the code store data structure.

Execution

Referring next to FIG. 3C, a flowchart of a method to be performed by a client computer according to an exemplary embodiment of the invention when a software package registered through the package manager is executed in the runtime environment is shown. This method is inclusive of the steps or acts required to be taken by the software package manager.

When the user requests execution of the software package, the runtime environment invokes the package manager (step 370) to locate the components necessary to run the software. The package manager matches the software package name to the corresponding entry in the code store data structure (step 371) to determine the directory, or directories, holding the components (step 373). In the embodiment in which the components are stored in an uncompressed zip file, or the like, the package manager uses the directory to find the particular components within the file. The package manager returns the location of the components to the runtime environment (step 375).

Uninstall

Finally, referring to FIG. 3D, a flowchart of a method to be performed by a client computer according to an exemplary embodiment of the invention when a software package registered through the package manager is uninstalled is shown. This method is inclusive of the steps or acts required to be taken by the software package manager.

When the user wants to uninstall a software package from the local computer, a standard uninstall routine provided by the runtime environment invokes the package manager

to update the code store data structure accordingly (step 380). The package manager removes the corresponding software package entry in the code store data structure (step 381). The package manager does not delete a component from the directory unless no other installed software package references it (step 383).

In one embodiment, the package manager scans every entry in the code store data structure to determine if another the entry for another software package references the local directory holding the component in question. In a first alternate embodiment, the package manager creates and maintains a tree structure of all shared components, so it can quickly determine if the component is shared with another software package. In a second alternate embodiment, the component is not deleted at the time the software package is uninstalled but instead a "garbage collection" routine is periodically run by the package manager. The garbage collection routine uses the code store data structure to determine if a component is referenced by any of the software packages installed on the local computer. Those components which are not referenced are then deleted.

Code Store Data Structure

Fig. 4 illustrates an exemplary embodiment of an entry in the code store data structure 400 suitable for use by the methods of the exemplary embodiments of the package manager described above. Each entry contains, at a minimum, five fields: a name field 401 for the distribution unit, a version field 403 for the distribution unit, a component field 405 that contains a list of the components in the distribution unit, a location field 407 that contains the location of the components on the client computer, and a source field 409 that contains a

pointer to the source of the distribution unit, such as a URL for a downloaded distribution unit. If a component is a platform-specific ("native code") file, such as a Microsoft Windows DLL, the file name is the component is stored in the component field. If a component is a package of Java classes, the component field contains the name of the Java package. In the case of a Java package, the code store entry has the following additional fields: a component version field 411 (which may be different from the version of the distribution unit; both are used by the package manager in resolving version dependencies), and a component type field 413 that indicates what type of Java classes the package contains, i.e., system, application, etc., both shown in phantom in FIG. 4 An optional data signature field 415, also shown in phantom, contains a digital signature affixed to the distribution unit, if it was signed. As well be familiar to one of skill in the art, the digital signature itself can contain security attributes about the package. The security attributes can be used by the package manager to prevent versions updates from a source other than the signer. Furthermore, a vendor may include software packages from third parties as dependencies in the distribution unit and the package manager uses the digital signatures on each dependent package to direct the corresponding components into different local directories.

In one embodiment of the code store data structure implemented in the Microsoft Windows environment, the system registry file is used as the repository for the code store entries and the package manager accesses the entries through the standard registry interface provided by the operating system.

Summary

The particular methods performed by the package manager of an exemplary embodiment of the invention have been described. The method performed by the package has been shown by reference to flowcharts including the steps from 300 to 355 during installation of a software package, the steps 370 to 375 during execution of a software package, and steps 380-385 during uninstallation of a software package. Additionally, an exemplary embodiment of a code store data structure has been described.

Open Software Description Implementation

In this section of the detailed description, a particular implementation of the invention is described that formats the manifest file using an Open Software Description (OSD) format. OSD specifies a vocabulary used for describing software packages and their dependencies for client computers which is a subset of the Extensible Markup Language (XML). XML is similar to the Hypertext Markup Language (HTML) used to write Web pages and provides a general method of representing structured data in the form of lexical trees. Using the XML model, markup tags in the OSD vocabulary are represented as elements of a tree. The three basic relationships between elements are "parent-of," "child-of," and "sibling-of." Distant relationships can be formed from recursive applications of the three basic ones. The basic XML elements that compose the OSD format are shown in Table 1 below. Additional information on the OSD vocabulary can be found in the Specification for the Open Software Description (OSD) Format published on August 11, 1997, and available for download from either Microsoft Corporation or Marimba, Inc.

Element	ABSTRACT
Content	<string>
Child of	SOFTPKG
Element	CODEBASE
Attributes	SIZE=<max-KB> -- the maximum allowable size for the software archive file. "Kilobytes" is the unit of measure. If SIZE is exceeded, then the software will not be downloaded. HREF=<URL> -- points to the archive to be downloaded. FILENAME=<string> -- specifies a file contained within the same archive as the OSD. If the OSD is used as a stand-alone file, then this attribute is ignored.
Child of	IMPLEMENTATION
Element	DEPENDENCY
Attributes	ACTION= (Assert | Install) -- Assert means: Ignore this SOFTPKG entirely if the dependency is not already present on the client's machine. Install means: If the dependency is not already present on the client's machine, go get it, then install it, so that the SOFTPKG has all the pieces it needs.
Child of	SOFTPKG, IMPLEMENTATION
Parent of	SOFTPKG
Element	DISKSIZE
Attributes	VALUE=<KB-number> -- approximate minimum number of bytes of disk space required by this implementation. "Kilobytes" is the unit of measure.
Child of	IMPLEMENTATION
Element	IMPLEMENTATION
Attributes	None Supported
Child of	SOFTPKG
Parent of	CODEBASE, DEPENDENCY, DISKSIZE, IMPLTYPE, LANGUAGE, OS, PROCESSOR, VM
Element	IMPLTYPE
Attribute	VALUE=<string> -- the type of the implementation.
Child of	IMPLEMENTATION
Element	LANGUAGE
Attributes	VALUE= <string> -- uses language codes as specified in ISO 639.
Child of	IMPLEMENTATION
Element	LICENSE
Attributes	HREF
Child of	SOFTPKG
Element	MEMSIZE
Attributes	VALUE = <KB-number> approximate minimum number of bytes of memory required by this implementation during execution. "Kilobytes" is the unit of measure.

Child of	IMPLEMENTATION
Element	OS
Attributes	VALUE= <string> -- see Appendix B for a list of possible values.
Child of	IMPLEMENTATION
Element	OSVERSION
Attributes	VALUE=<string>
Child of	OS
Element	PROCESSOR
Attributes	VALUE= <string> -- see Appendix B for a list of possible values.
Child of	IMPLEMENTATION
Element	SOFTPKG
Attributes	HREF -- indicates the Web page associated with a software distribution. Optional. NAME=<string> -- the name of the distribution. For a given SOFTPKG, this attribute should be a unique identifier. The client can use NAME to distinguish one SOFTPKG from all others. A software package's "friendly-name" is specified by using the TITLE element. VERSION=<string>
Child of	DEPENDENCY
Parent of	ABSTRACT, CODEBASE, IMPLEMENTATION, DEPENDENCY, TITLE
Element	TITLE
Content	<string>
Child of	SOFTPKG
Element	VM
Attributes	VALUE
Child of	IMPLEMENTATION

Table 1

The OSD vocabulary can be used in a stand-alone XML manifest file to declare the dependencies between different software components for different operating systems and languages. The OSD file provides instructions that can be used to locate and install only the required software components depending on the configuration of the target machine and what software is already present. The OSD formatted manifest file also can be embedded in an archive file, such as a Java Archive (.JAR) file, or a composite, compressed file, such as a

cabinet (.CAB) file, that contains the component's distribution unit to form a distribution unit file.

Additionally, the manifest file can specify an XML tag, "namespace," that causes the package manager to isolate software packages from one another even if the same component is used by multiple packages:

```
<JAVA>
. . . .
<NameSpace>Fred's Software Company</NameSpace>
```

Thus the use of namespaces avoids version mismatches among software packages.

A namespace is analogous to a directory in a file system, such as implemented in the Windows family of operating systems, in that installing applications in different directories provides isolation for the applications. Previously a namespace was global to all applications installed on a system so that all files and components in the namespace were accessible by the applications. The global nature of previous namespaces presents difficulties in naming files and components because an application programmer has to avoid common names to prevent potential conflicts with identically named files and components for another application installed in the same namespace. Installing the second of the two applications would likely cause one or both applications to fail, just as placing all files for all applications installed on a computer into a single directory frequently causes conflicts between the applications.

In the present invention, the presence of a namespace XML tag in the manifest file causes the package manager to associate the files and components of the corresponding application in the code store data structure with the unique namespace specified in the tag.

When an application is executed, the package manager passes the associated namespace name to the computer's runtime environment so that any files and components installed in that namespace are visible to the application while files and components installed in other namespaces are not. Using the example XML tag above, Fred's CoolestApp is associated with a namespace called "Fred's Software Company," would execute in the "Fred's Software Company" namespace, and have access to any files or components installed in the "Fred's Software Company" namespace. Similarly, an XML tag for Bob's identically named "CoolestApp" would specify "Bob's Software Company" as the namespace, execute in the "Bob's Software Company" namespace, and have access to any files or components installed in the "Bob's Software Company" namespace. Neither Bob's CoolestApp nor Fred's CoolestApp can access a common component or file installed in the other's namespace. Therefore, because of the isolation that namespaces provide, both Fred and Bob are assured their applications will function correctly even though identically named and having common components or files, and that the applications will continue to function correctly irregardless of the number of CoolestApps using the same components or file which may be installed on the computer.

Continuing with the distribution of Fred's Software Company's CoolestApp, the manifest file in a first exemplary embodiment in this section is stored separately from the distribution unit at http://www.fsc.com/coolestapp.osd. The corresponding distribution unit is stored in a cabinet file at http://www.fsc.org/coolestapp.cab. The CoolestApp's dependency on the components of the earlier CoolApp is indicated with a "DEPENDENCY" tag and refers the package manager to the CoolApp manifest file at http://www.fsc.org/coolapp.osd (not

shown). The CoolApp manifest file directs the package manager to the location of the distribution unit for the CoolApp.

```
<SOFTPKG NAME="com.fsc.www.coolestapp" VERSION="1,0,0,0">
    <TITLE>CoolestApp</TITLE>
    <ABSTRACT>CoolestApp by Fred's Software Company</ABSTRACT>
    <LICENSE HREF="http://www.fsc.com/coolestapp/license.html" />

    <!—FSC's CoolestApp is implemented in native code -->
    <IMPLEMENTATION>
      <OS VALUE="WinNT"><OSVERSION VALUE="4,0,0,0"/></OS>
      <OS VALUE="Win95"/>
      <PROCESSOR VALUE="x86" />
      <LANGUAGE VALUE="en" />
      <CODEBASE HREF="http://www.fsc.org/coolestapp.cab" />

      <!—CoolestApp needs CoolerApp -->
      <DEPENDENCY>
        <CODEBASE HREF="http://www.fsc.org/coolapp.osd" />
      </DEPENDENCY>
    </IMPLEMENTATION>
</SOFTPKG>
```

Had the CoolApp manifest file been stored in a cabinet distribution unit file along with the CoolApp components, the location of the distribution unit file would have been http://www.fsc.org/coolapp.cab.

In a second exemplary embodiment of the invention for purposes of this section, components contained in a distribution unit file are caused to be installed by OSD tags embedded on a Web page. If Fred's Software Company's Web page requires additional software to be downloaded and installed for viewing the page, FSC can use the OSD vocabulary within HTML commands to have the user's browser download the necessary components as shown in the two examples below.

```
<OBJECT CLASSID="clsid:9DBAFCCF-592F-101B-85CE-00608CEC297B"
```

```
        VERSION="1,0,0,0"
        CODEBASE=" http://www.fsc.com/coolestapp.osd "
        HEIGHT=100 WIDTH=200 >
</OBJECT>
```

-or-

```
<APPLET code=myapplet.class id=coolestapp width=320 height=240>
        <PARAM NAME=useslibrary VALUE="coolestapp">
        <PARAM NAME=useslibraryversion VALUE="1,0,0,0">
        <PARAM NAME=useslibrarycodebase VALUE="http://www.fsc.com/coolestapp.osd
">
  </APPLET>
```

The HTML <OBJECT> or <APPLET> tag informs an OSD-aware client browser, such as Microsoft Explorer 4, that there is additional software required to view the Web page. The browser invokes the package manager to execute the software package if it is already installed or to install it if not. If not already installed, the package manager instructs the browser to download the distribution file unit and proceeds with the installation as described in the previous section. The "CODEBASE" element in <OBJECT> and the "useslibrarycodebase" tag in <APPLET> can point to the manifest file or to the distribution unit file.

In a third exemplary embodiment of the invention for purposes of this section, a distribution unit file is used to automatically distribute software from Fred's Software Company's server to the user's computer. This automatic distribution across a network employs "channels" to which the user subscribes to automatically "push" software components through a client agent such as a browser. The channel is described using a Channel Definition Format (CDF) which is also based on XML. A CDF file uses the OSD elements to inform a CDF-aware client agent as to what software components should be downloaded and installed.

```
<CHANNEL HREF="http://www.fsc.com.intropage.htm">
 <SELF="http://www.fsc.com/software.cdf" />
 <TITLE>A Software Distribution Channel</TITLE>

 <SOFTPKG
      HREF="http://www.fsc.com/aboutsoftware.htm"
      AUTOINSTALL="yes"
      NAME="{D27CDB6E-AE6D-11CF-96B8-444553540000}"
      VERSION="1,0,0,0">

   <IMPLEMENTATION>
     <OS VALUE="WinNT"><OSVERSION VALUE="4,0,0,0"/></OS>
     <OS VALUE="Win95"/>
     <PROCESSOR VALUE="x86" />
     <CODEBASE HREF="http://www.fsc.com/coolestapp.cab" />
   </IMPLEMENTATION>
</SOFTPKG>
  </CHANNEL>
```

This section has described a particular implementation of the package manager which is directed to install software by OSD elements embedded in an XML document. The processing of a manifest file described in previous section when written as XML document is described. In addition, alternate embodiments in which a separate XML document resides on a Web page to direct a browser to invoke the package manager to install a software package is also described in this section.

Conclusion

A software package manager has been described which manages the installation, execution and uninstallation of software packages acquired through various media. The software manager uses a manifest file, a distribution unit, and a code store data structure to accomplish its functions. Although specific embodiments have been illustrated and described

herein, it will be appreciated by those of ordinary skill in the art that any arrangement which is calculated to achieve the same purpose may be substituted for the specific embodiments shown. This application is intended to cover any adaptations or variations of the present invention.

For example, those of ordinary skill within the art will appreciate that the file and data structures described herein can be easily adapted to future distribution media. Furthermore, those of ordinary skill within the art will appreciate that future extensible languages which are platform and operating system independent can be used to direct the software package managers actions.

The terminology used in this application with respect to is meant to include all hardware and software platforms. Therefore, it is manifestly intended that this invention be limited only by the following claims and equivalents thereof.